Introduction to Compiler

By:
Trusha R. Patel
Asst. Prof.
CE Dept., CSPIT, CHARUSAT

Basic

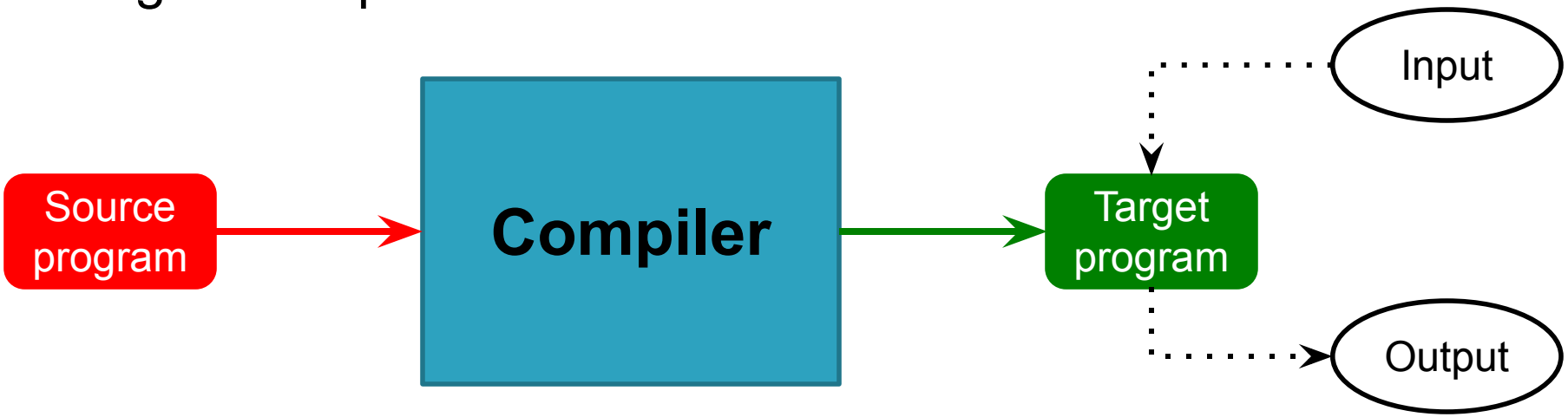
- Software running on all computers was written in some programming language
- Before a program can run, it first must be translated into a form in which it can be executed by a computer
- This translation is done by compiler

Compiler

- Language processor

- It is a program reads a program written in source language and translates it into an equivalent program in target language Its main role is to report errors during translation process

- It target program is executable machine code then it take input and gives output



Interpreter

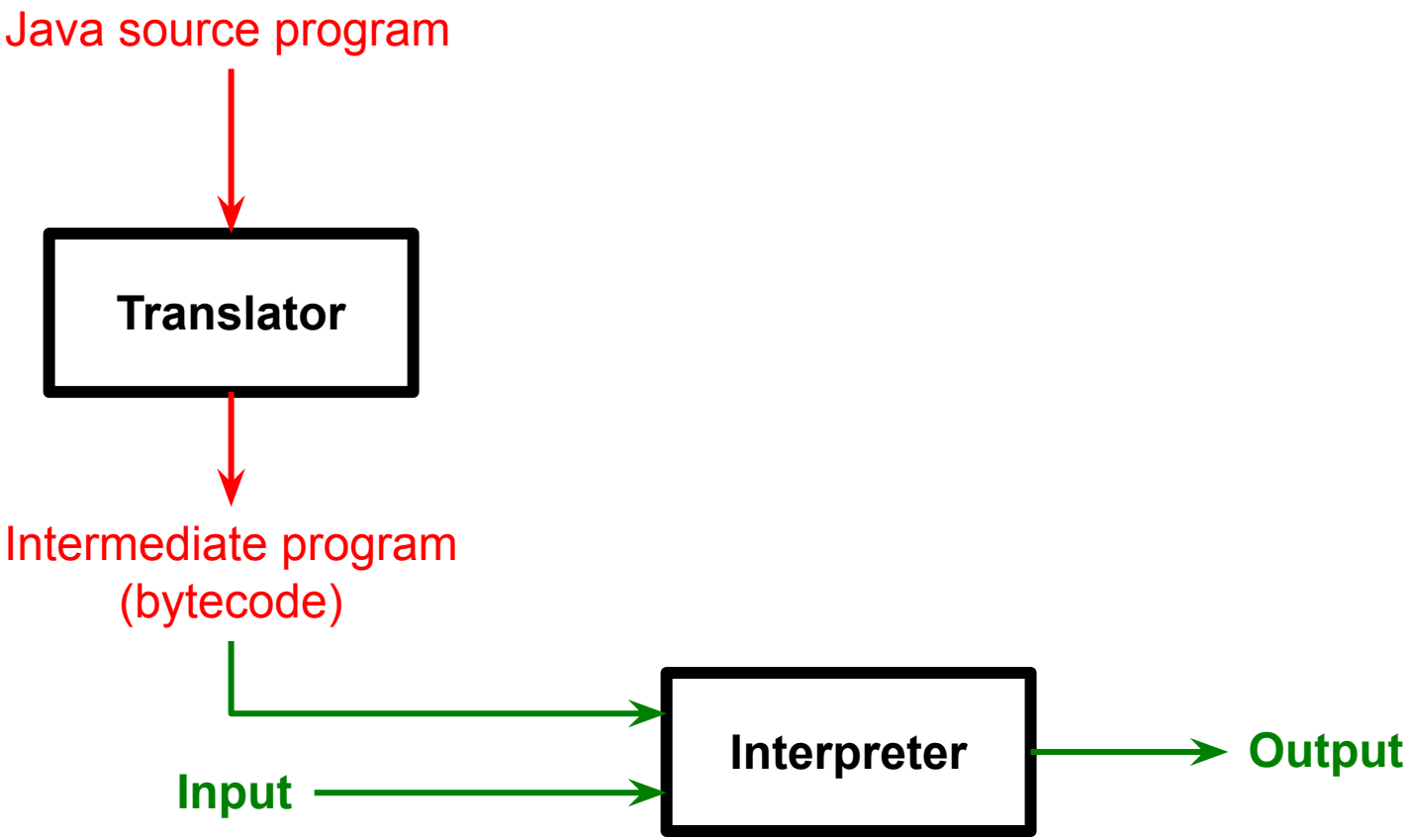
- Language processor
- Not produce target program as result like translator
- Directly execute the operations specified in the source program on input

Compiler vs. Interpreter

- Compiler
  - Machine-language target program generated by a compiler is usually much faster at mapping input to output
- Interpreter
  - Interpreter usually give better error diagnostics

Java Language Processor

- Combines compilation and interpretation



Language Processors

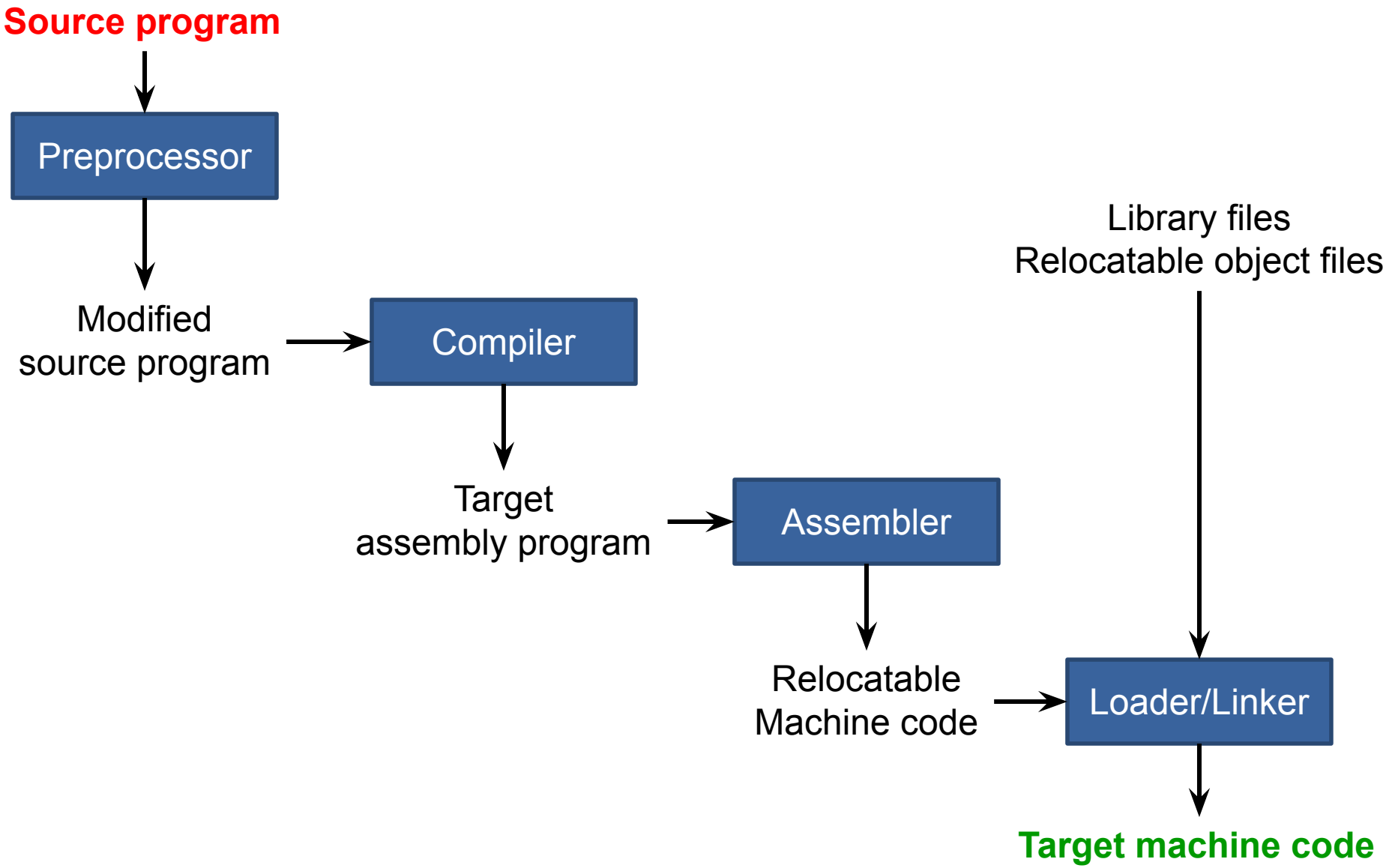
- Source program may be divided into modules (stored in different files)
- Task of collecting source program and expansion of shorthand (macro) is done by Preprocessor
- Modified program is given to Compiler that produce an assembly-language program as output
- Assembly-language program is processed by Assembler and generate relocatable machine code

Language Processors

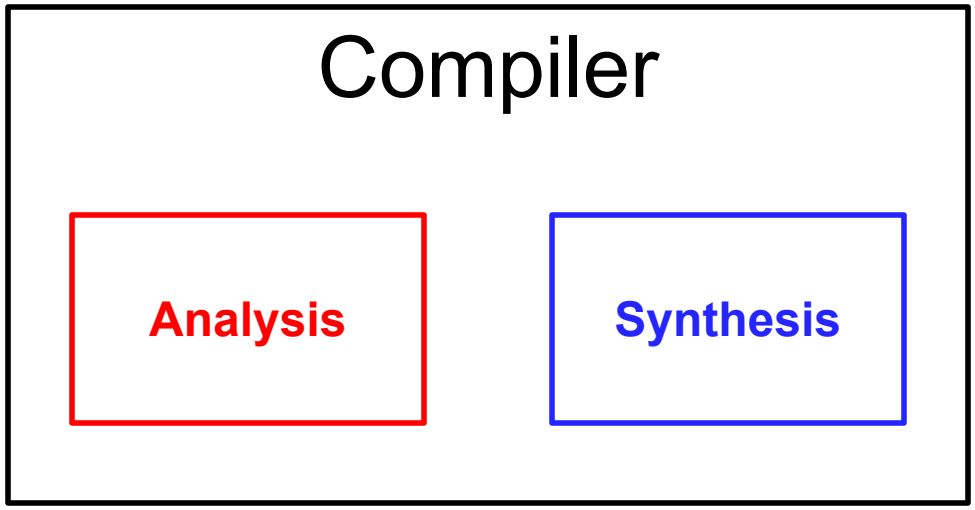
- Large programs are often compiled in pieces, so relocatable machine code may have to be linked together which is done by Linker
- It resolves external memory address

- Loader puts all of executable object files into memory for execution

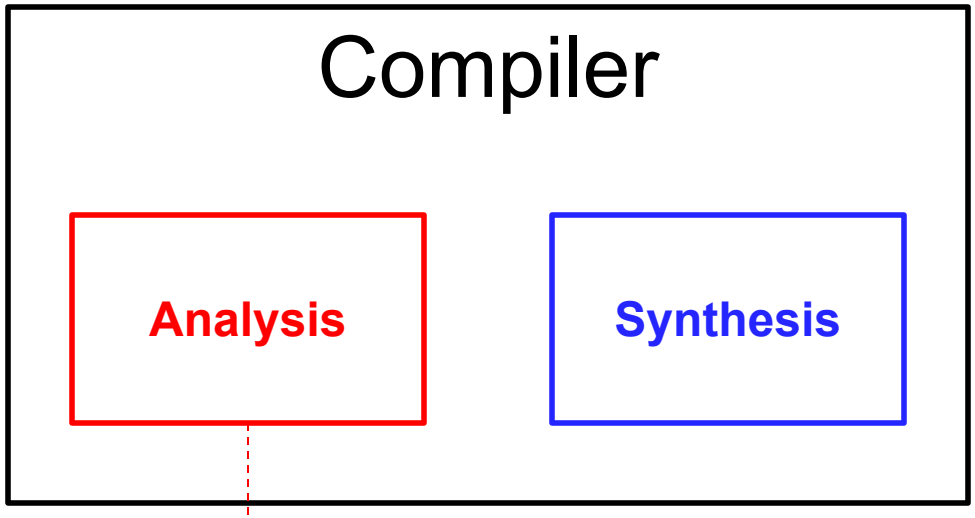
Language Processors



Structure of Compiler

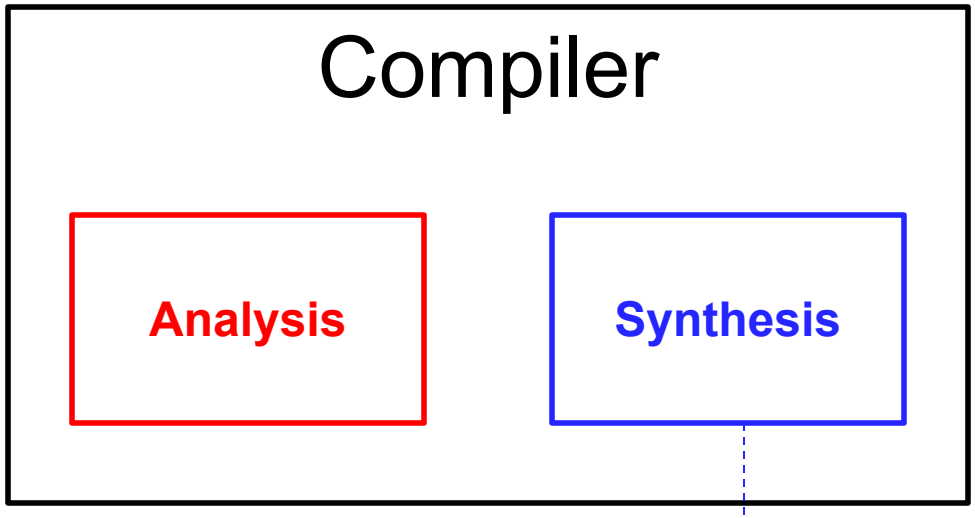


Structure of Compiler



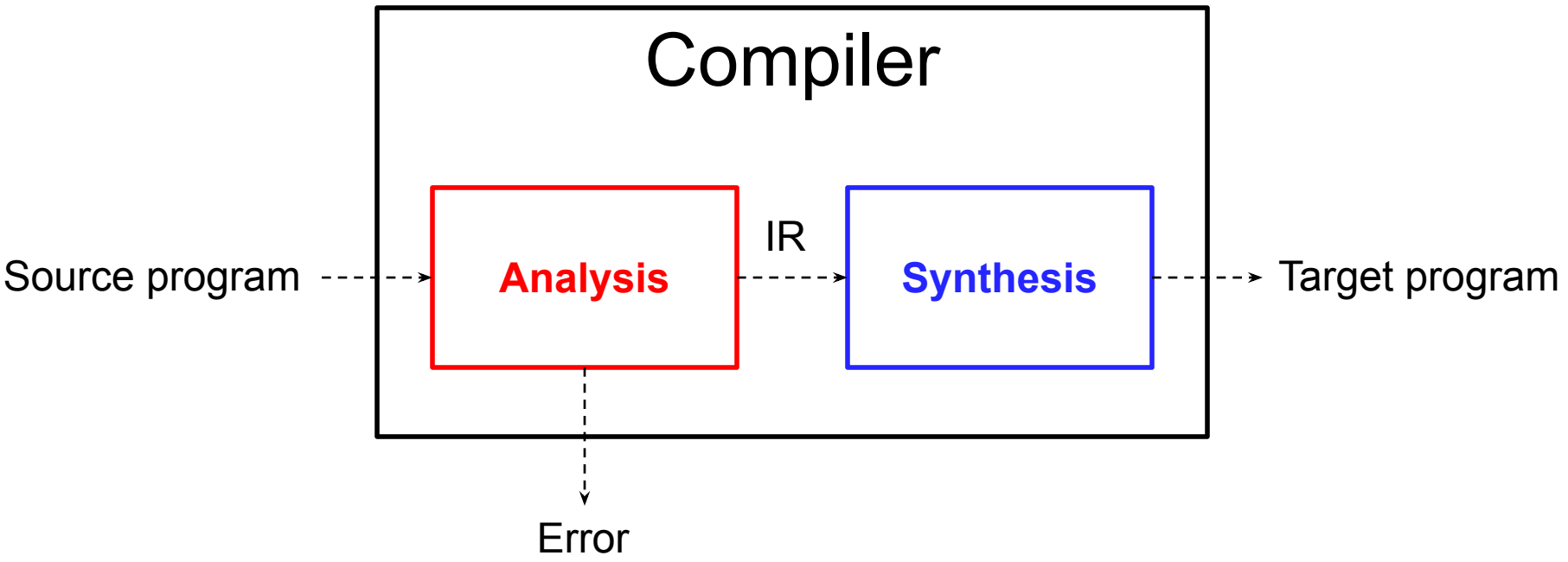
- Break source program into pieces
- Check grammatical structure
- Give error (if any)
- Store information in symbol table
- Generate intermediate representation (IR)

Structure of Compiler

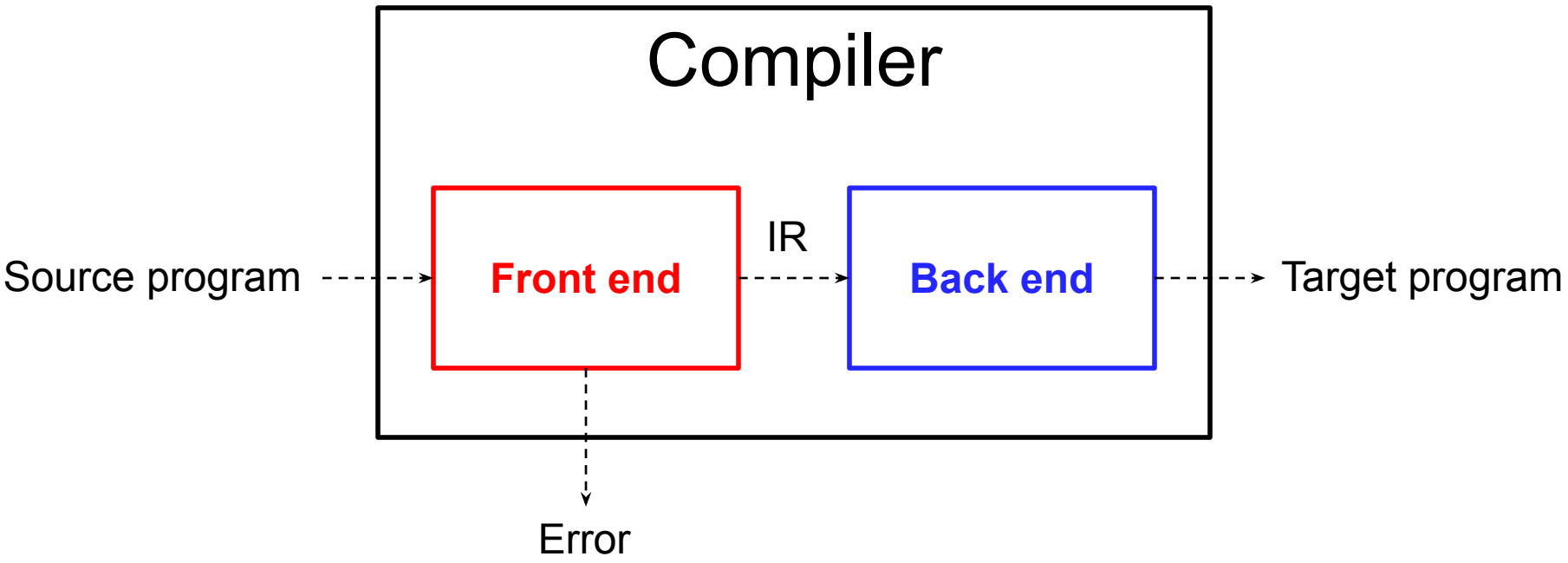


- Construct target program from IR and information in Symbol table

Structure of Compiler



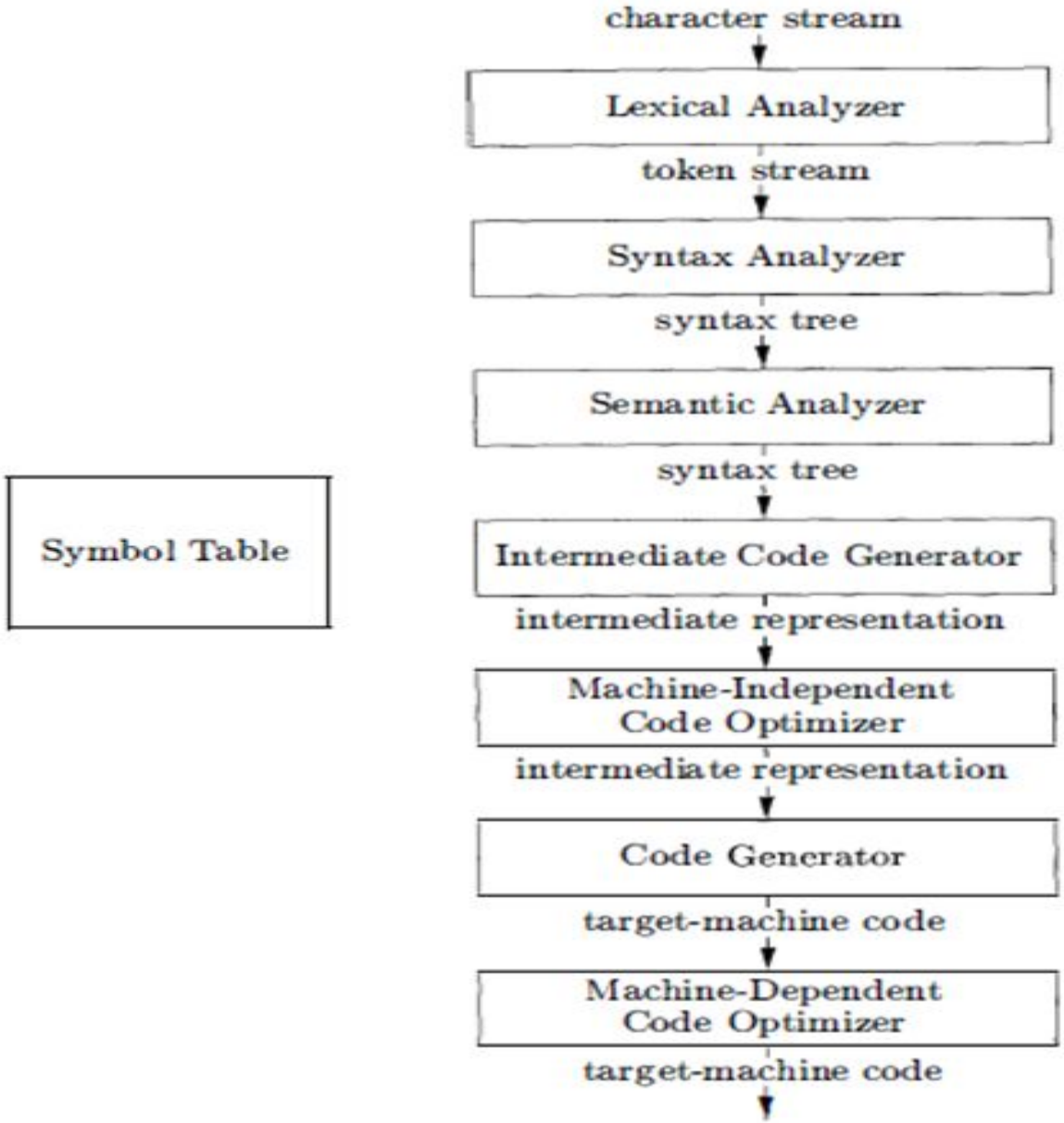
Structure of Compiler



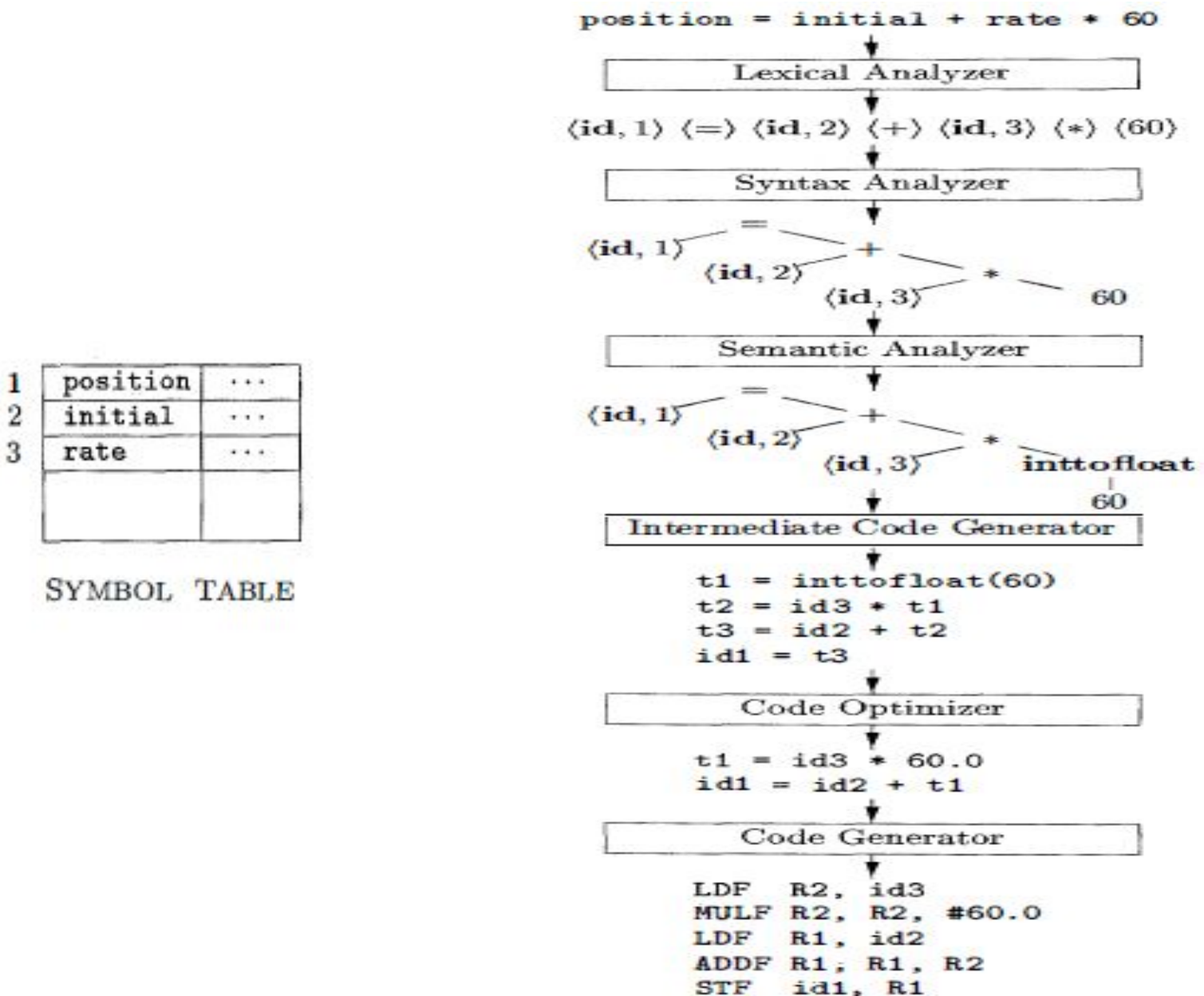
Structure of Compiler

- Phase
  - Compiler operates as a sequence of individual logical unit called phase, each of which translates one representation of the source program to another.

Phases of Compiler



Translation of Assignment Statement



Phase-1 : Lexical Analysis

- Lexical analysis is also called scanning

- Reads the stream of characters and group the characters into meaningful sequence called lexemes and for each lexeme produce token in given form

  < token-name , attribute-value>

  pointer to an entry in symbol table for token

  abstract symbol that is used during syntax analysis

Phase-2 : Syntax Analysis

- Syntax analysis is also called parsing
- Creates tree-like intermediate representation that determines the grammatical structure of the token
- Typical representation is syntax tree in which interior node represents an operation and the children of the nodes represents the arguments of the operation

Phase-3 : Semantic Analysis

- Uses syntax tree and information in symbol table
- Check the source program for semantic consistency
- Important part of semantic analysis is type checking
- Some language may support coercion
  (e.g. internal type casting in C)

Phase-4 : Intermediate Code Generation

- After syntax and semantic analysis many compiler generate an explicit low-level or machine-like intermediate representation (IR)

- Two important properties of IR
  - Should be easy to produce
  - Should be easy to translate into target machine code

Phase-5 : Code Optimization

- It attempts to improve the intermediate code so that better target code will result

- Generate shorter IR, which can convert faster in target code

Phase-6 : Code Generation

- Takes IR as input and maps it into the target language

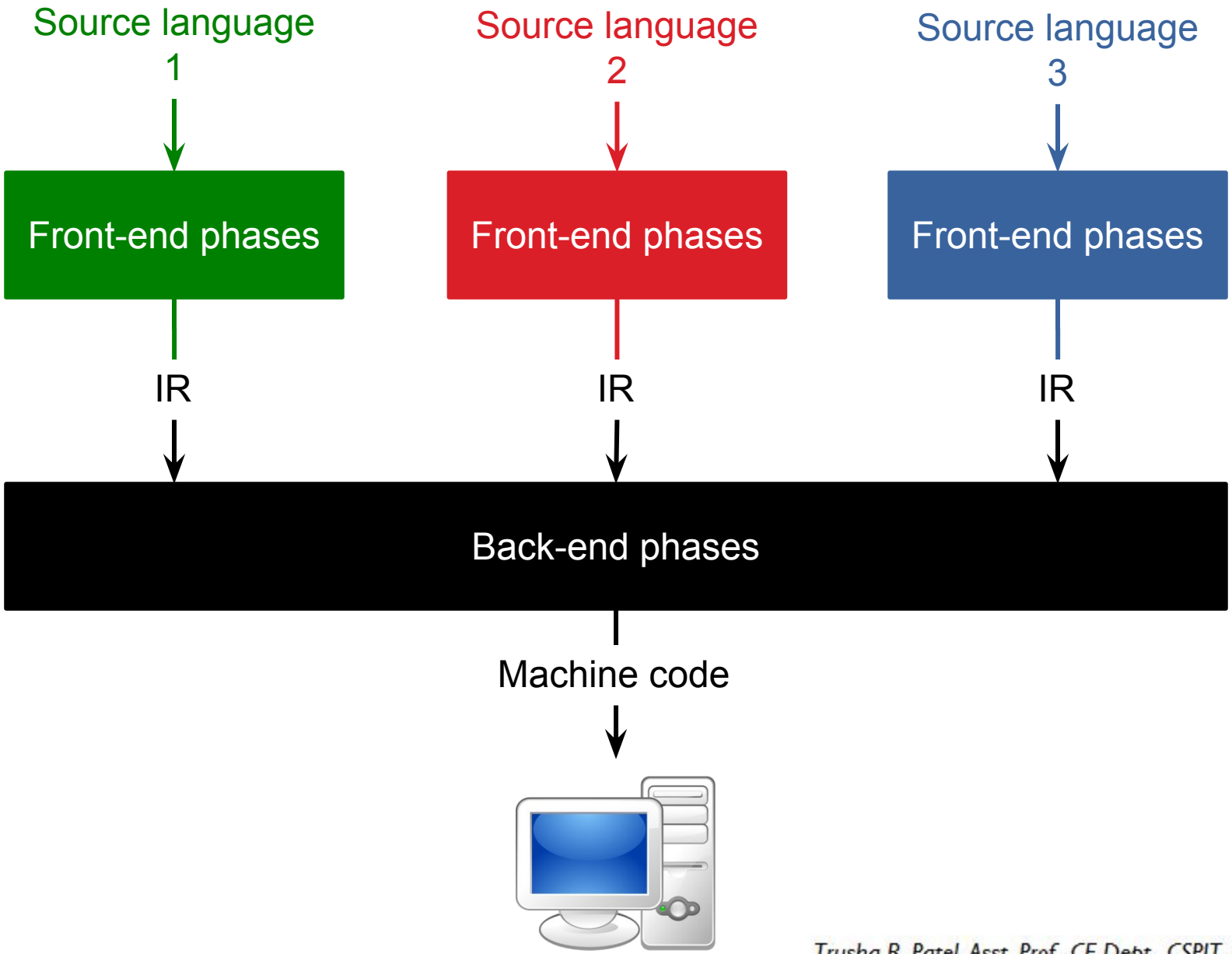
Grouping of Phases

- Phases deals with the logical organization of a compiler
- Phases may be grouped together into a pass that reads an input file and generate an output file
- Front-end phases (lexical analysis, syntax analysis, semantic analysis, intermediate code generator) might be grouped together into one pass
  Back-end phases (code optimizer, code generator) are grouped into second pass

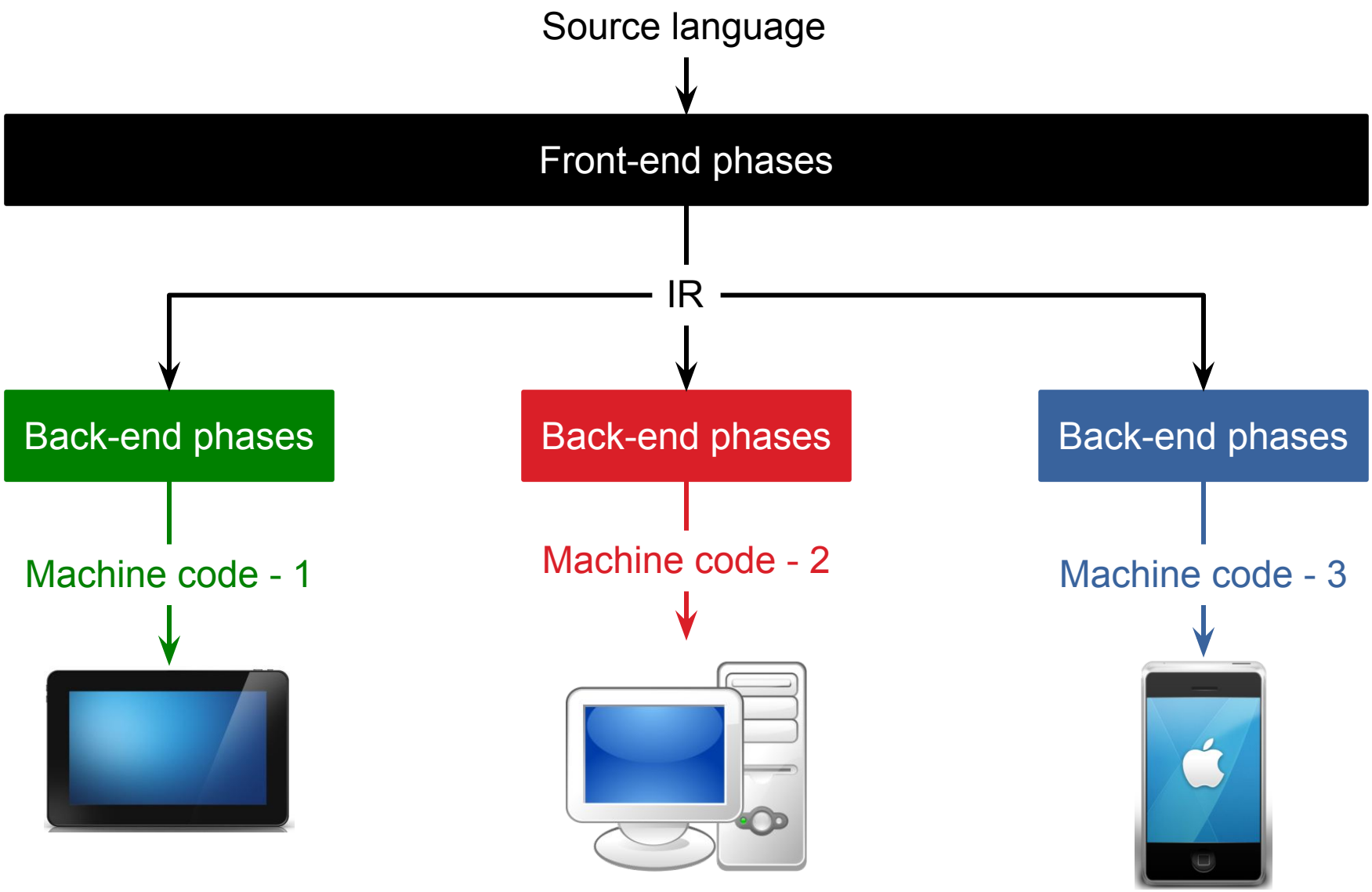
Grouping of Phases

- Front-end phases : independent of machine
  dependent on language
- Back-end phases : dependent on machine
- Intermediate representation (IR) provides interface between front-end and back-end
- Which this collection it is possible to produce compiler for different source language for one target machine

Grouping of Phases



Grouping of Phases



Compiler Construction Tools

1. Parser generators
   - Automatically produce syntax analyzers from grammatical description
2. Scanner generators
   - Produce lexical analyzers from regular expression description
3. Syntax-directed translation engines
   - Produce collections of routines from parse tree and generate intermediate code

Compiler Construction Tools

4. Code-generator generators
   - Produce code generator from collection of rules for translating each operation of intermediate language into machine language for target machine

5. Data-flow analysis engines
   - Gather information about how data values are transmitted from on part of the program to other part

6. Compiler-construction toolkits
   - Provide integrated set of routines for constructing various phases of compiler

END